CHAPTER

# NINETEEN

# CONTRASTS AND COMPARISONS

The systems discussed in this book span the important ideas for coordinated computing. The variety of mechanisms and structures in these systems may seem staggering. But as we argued in Chapter 5, most of the important differences can be described as choices in a decision space. This chapter has three sections. In the first, we review the choice dimensions and describe where each system lies in the decision space. We summarize these results in a series of tables. The second section discusses specific common themes in greater detail. Often systems are refinements of earlier ideas; this section conveys some of this historical development. In the last section we propose a unified model, showing how an appropriate set of primitives is sufficient to achieve any of the behaviors.

## 19-1 GENERAL COMPARISONS

### Dimensions of Distribution

In Chapter 5, we identified 12 key dimensions in the design of coordinated computing systems. In this section, we review these dimensions, ascribing to each system its place in the dimension space.* The differences among systems are best summarized in tabular form. Unfortunately, a table of 12 dimensions and 18 systems would be incomprehensible. Instead, we have broken our table into three parts: (1) Table 19-1 examines the general goals and structure of each system, (2) Table 19-2 exhibits aspects of intrasystem communication, and (3) Table 19-3 presents the systems with respect to the remaining dimensions.

* Other authors who have pursued the theme of comparisons of concurrent and distributed languages include Andrews and Schneider [Andrews 83], Mohan [Mohan 80], Rao [Rao 80], and Stotts [Stotts 82].

**Problem Domain (A)** The systems described in this book range from simple models to elaborate languages. Much of this diversity arises from differences in task domains; the different systems are meant to be solutions to different problems. The models tend to be mathematical. For example, Shared Variables is a tool for studying algorithmic analysis and program correctness; Concurrent Processes, program semantics; and Petri Nets, system correctness. The programming languages focus on the problems of writing programs. They divide into the "general-purpose" or *pragmatic* languages (PLITS), programming languages directed at implementing operating systems (Concurrent Pascal, Distributed Processes, SR, and Cell), and languages concerned with the issues of particular subproblems of operating systems (Ada, embedded systems; and Argus, distributed databases). The heuristic systems (Distributed Hearsay-II, Contract Nets, Ether) focus on organizing distributed problem solving.

Some systems assume an intermediate point between these extremes: CSP is not only a language for systems implementation but also a model of both hardware and program semantics; Actors and IAP strive both to have a well defined semantics and to be pragmatic programming environments. And lastly, Exchange Functions, while in many ways similar to the formal Concurrent Processes, is directed toward a completely different goal— requirements specification for program development.

Of course, we mean these distinctions to capture the thrust of a system design, not the minor intentions. Languages for systems implementation must be concerned with the pragmatic problems of programming. Accompanying the design of many new programming languages is an attempt to provide their formal semantics. And good models often correspond to the reality of building the kinds of systems they model. Nevertheless, understanding the differing intentions of the system designers is an essential component for understanding the diversity in the designs.

**Explicit Processes (B)** These systems all support concurrent computation. One issue for concurrent systems is designating which actions can be executed concurrently. Almost all our systems use explicit processes to show concurrency. In *explicit process systems*, the programmer causes process entities to be created. Once created, these processes compute concurrently (subject to various synchronization constraints). Most systems call their processes, "processes." Occasionally, a system uses another name for the same concept: tasks in Ada, modules in PLITS, actors in Actors, cells in Cell, and sprites in Ether.

Despite the popularity of processes, not all systems use them. Four systems indicate concurrency without processes. In IAP, each call to cons and frons creates a suspension (a form of task) to be resolved by the underlying system. This task will be satisfied either lazily, by otherwise idle resources, or immediately, driven by a demand for its answer. Distributed Hearsay-II combines the pattern-directed invocation of productions with the broadcast communication of blackboards. And Data Flow and Petri Nets abstract concurrency by the independent flow of graph tokens.

For the sake of comparison, we treat concurrency in IAP as the concurrency of the underlying evaluators (communicating through the realization of cell values), concurrency in Distributed Hearsay-II as the concurrency of knowledge sources, and concurrency in Petri Nets and Data Flow as the concurrency of the transitions and actors communicating with tokens.

**Process Dynamics (C)** Process dynamics describes the change in number and variety of processes through the execution of a program. Some systems allow programs to create new processes during execution. Others require that all processes be defined at system creation (*static processes*). Shared Variables, Exchange Functions, Petri Nets, Data Flow, CSP, Concurrent Pascal, Distributed Processes, SR, Distributed Hearsay-II, and Contract Nets are static process systems.

Two different syntactic mechanisms support dynamic process creation, explicit allocation (*dynamic*) and lexical program elaboration (*lexical*). Systems with explicit allocation have a statement to create a new process, much as Pascal has a statement to allocate a new record. Actors, IAP, Ada, and PLITS have explicit process allocation. Lexical elaboration creates processes by combining declarations with recursive program structures. That is, if procedure P declares process A and then calls itself recursively, the recursive invocation of P creates another copy of A. Concurrent Processes, Ada, PLITS, Cell, Argus, and Ether create new processes by lexical elaboration.

By and large, pragmatic systems more frequently allow dynamic process creation than systems-oriented languages. Systems based on compilation restrict new processes to be described by a "type," though processes may be specialized by parameterization. Some interpreted systems (such as Actors) allow the description of a process to be generated at the time the process is created.

Systems with explicit allocation ought to provide names (pointers) to the newly created processes. These names can then be passed between processes, allowing other processes to become aware of the new processes and send them messages. Lexical process creation often restricts interprocess communication to lexical scopes.

**Synchronization (D)** The differences that distinguish concurrent computation from simple serial processing center on communication and synchronization among processes. The first notable division between systems is the issue of synchronous versus asynchronous communication. In synchronous communication, both processes "attend" to the communication. Every communication request is matched by a reception; a process cannot send a second communication until the first has been handled. In asynchronous communication, processes send requests without regard to their reception; a process is free to send a request and continue computing.

Ten systems are asynchronous: Shared Variables, Petri Nets, Data Flow, Actors, IAP, Concurrent Pascal, PLITS, Distributed Hearsay-II, Contract Nets, and Ether. SR provides both synchronous and asynchronous communication. The other systems are synchronous.

**Table 19-1 Goals and Structures**

| System | (A) Task domain | (B) Explicit processes | (C) Process dynamics |
|---|---|---|---|
| **Shared Variables** *Lynch & Fischer* | Analysis, correctness | Processes | Static |
| **Exchange Functions** *Fitzwater & Zave* | Requirements specification | Processes | Static |
| **Concurrent Processes** *Milne & Milner* | Semantics | Processes | Lexical |
| **Petri Nets** *Petri* | Correctness, modeling | Structure | Static |
| **Data Flow** *Dennis/Arvind & Gostelow* | Pragmatics | Structure | Static |
| **CSP** *Hoare* | Systems, semantics | Processes | Static |
| **Actors** **Hewitt** | Pragmatics, semantics | Processes | Dynamic |
| **IAP** *Friedman & Wise* | Pragmatics, semantics | Tasks | Dynamic |
| **Concurrent Pascal** *Brinch Hansen* | Systems | Processes | Static |
| **Dist. Processes** *Brinch Hansen* | Systems | Processes | Static |
| **Ada** *DoD* | Systems (embedded) | Processes | Dynamic, lexical |
| **PLITS** *Feldman* | Pragmatics | Processes | Dynamic, lexical |
| **SR** *Andrews* | Systems | Processes | Static |
| **Cell** *Silberschatz* | Systems | Processes | Lexical |
| **Argus** *Liskov* | Systems (database) | Processes | Dynamic |
| **Dist. Hearsay-II** *Lesser & Corkill* | Problem solving | Productions | Static |
| **Contract Nets** *Smith & Davis* | Problem solving | Processes | Static |
| **Ether** *Kornfeld & Hewitt* | Problem solving | Processes | Dynamic |

**Buffering (E)** A system's buffer size is the number of messages from a given process that can be pending at one time. The interesting distinction is between systems with *bounded* buffers and systems with *unbounded* buffers—that is, systems for which a process's pending messages can occupy only a limited space in contrast with systems that allow a process to send an unbounded number of messages. Synchronous systems invariably have bounded buffering—a synchronous process can have only a finite number of pending messages.* Message-based, asynchronous systems (Petri Nets, Actors, PLITS, SR, and Contract Nets) have unbounded buffering, as do pattern-invocation systems (Distributed Hearsay-II and Ether). One version of Data Flow uses bounded buffers [Dennis 74], another unbounded [Arvind 77]. Additionally, some systems (Shared Variables, IAP, Concurrent Pascal, Ada, and SR) permit processes to share memory. Shared memory is a form of bounded, asynchronous communication.

**Information Flow (F)** Interprocess communication is the transfer of information. Clearly, the mere occurrence of a communication is informative. Thus (particularly in synchronous systems), communication provides a synchronization signal. However, it is usually desirable to transmit more than pure synchronization—effectively, to transmit information. The various systems propose different organizations for information flow. For some, communication provides a sender and a receiver; the information flow is unidirectional (*uni*) from the sender to the receiver. Of course, unidirectional communication is a consequence of message-based asynchronous systems (Petri Nets, Data Flow, Actors, PLITS, Contract Nets, and the asynchronous primitive in SR). Only a single synchronous system (CSP) has unidirectional information flow.

Shared memory introduces its own nuances of information flow. Shared memory is naturally asynchronous. In its simplest form, like IAP, Distributed Hearsay-II, and Ether, it is unidirectional, from a "writer" to a "reader." Monitors in Concurrent Pascal structure the exchange to be bidirectional—first, a transfer of information to the monitor and then a reception of return values. In Shared Variables, the exchange is (for read-write processes) somewhat the opposite—first a value is received from the variable and then the variable is set to a new state.

Synchronous communication can be treated either as a single, instantaneous event or as an event that progresses through stages. The first of these is simultaneous, bidirectional communication (*bi-sim*). Exchange Functions and Concurrent Processes have simultaneous, bidirectional communication. The synchronous programming languages have delayed bidirectional communication (*bi-del*). In this organization, processes enter a rendezvous. In *rendezvous*, the requesting process transfers information to the called process. This process then computes

* Naively, it would seem that a single-message buffer would suffice for synchronous systems. However, systems with output guards can have a pending message for each guarded output clause.

a response and sends that response back to the original caller. This pattern, the remote procedure call, is a generalization of the procedure call, with its stepwise transfer of query and control to the called procedure, the execution of the body of the procedure, and the return of answers and control to the caller. Distributed Processes, Ada, SR, Cell, and Argus use delayed bidirectional communication.

Strangely, none of the models or languages has generalized the coroutine as a communication model. Such a generalization would provide first an establishment of communication and then an alternation of information transfer and computation through the entire transaction. However, processes are adequate coroutines of and by themselves; in any system that provides some filtering of message reception it is straightforward to obtain this control pattern.

**Communication Control (G)** The various models and languages show the most diversity in mechanisms for establishing communication. To handle this variety, we break this dimension into two parts, control and connection. Control concerns the actions that processes take to communicate, including the facilities they have for choosing a communication partner. Connection is an issue of naming: to what does a communicating process refer?

Seven systems (Shared Variables, Exchange Functions, Concurrent Processes, CSP, IAP, Concurrent Pascal, and Distributed Hearsay-II) treat communicators as equals (*equal*). In Exchange Functions and Concurrent Processes, communicating processes take identical actions. Exchange Functions allows processes to test whether communication is available and to abort a communication attempt if it is not (*immed. time-out*). CSP treats processes (roughly) as equals. It introduces asymmetry with unidirectional information flow. Input guards (and, extensionally, output guards) provide further concurrency control (*I/O guards*). In Shared Variables, IAP, and Distributed Hearsay-II, communication is anonymous and nondirective. Processes asynchronously read and write shared storage. No process has, a priori, a distinguished role. In Concurrent Pascal, processes communicate through shared monitors. Processes do not address each other directly; hence, communicating processes are equals.

The other systems specify roles for the "calling" and "called" processes. Petri Nets, Data Flow, Actors, Distributed Processes, Argus, and Ether treat the called process as a passive server (*passive*) that accepts requests without controlling the order of their reception. (However, Distributed Processes programs can then use guarded commands to order accepted requests.) Ada, PLITS, SR, Cell, and Contract Nets allow the called process some freedom in choosing which requests to serve (*active*). All segregate requests into groups. In Ada, SR, and Cell, requests are grouped by entry queues; in PLITS, by transaction keys. All but PLITS have input guards to read from one of several queues at once (*i-guard*); all have functions to determine if a particular queue is empty.

Additionally, some languages add their own features for communication control. PLITS, SR, and Cell allow filtering of requests by origin (*send-filt*); SR and Cell, ordering requests by priority (*priority*); and SR, guards that can examine not only the internal state of the process, but also the message (*mess-grd*). Ada allows both input and output guards and permits time-outs by both the calling and called processes. Concurrent Pascal and Cell can suspend processing a request and then later resume it (*suspend*).

**Connection (H)** These systems use four different syntactic forms to channel communication: ports, names, entries, and pattern-selective broadcast. These syntactic devices can be used by the sender, the receiver, or both the sender and the receiver of the communication.

Communication through a symbol external to communicating processes is communication through a port (*port*). Shared memory systems (Shared Variables, IAP, and Concurrent Pascal) use ports, where the shared memory is the port. Additionally, Exchange Functions and Concurrent Processes use explicit ports. We view the shared places of Petri Nets as another form of port. Data Flow, Actors, and PLITS direct communication at an unmodified process (*name*). None of these systems requires the receiver of a communication to describe the sender. However, PLITS processes can filter requests by transaction keys (*key-filt*), using the transaction keys as a kind of entry mechanism.

Several of the languages (Distributed Processes, Ada, SR, Cell, and Argus) focus communication on an entry (*entry*) in the called process. In Ada, SR, and Cell, a called process can have several entries and accept requests from them in an order determined by program control. In Distributed Processes and Argus, entries are not explicitly referenced by the recipient program. CSP pattern matching serves a similar filtering purpose.

Three heuristic systems, Distributed Hearsay-II, Contract Nets, and Ether, specify subproblems as tasks and distribute these tasks in a broadcastlike fashion. On the basis of the pattern-described interests of the processing elements, the system directs relevant messages to them.

**Time (I)** Einstein asserted that time is relative. Relativity arises because the information of an event cannot travel faster than the speed of light. Lamport has argued that a similar principle applies to distributed computing systems [Lamport 78]. In a distributed system, it is meaningless to refer to the absolute "time" at which an event happened, particularly from the perspective of a single process. Instead, the information of an event diffuses, through communication, to interested processes. Processes can only be synchronized, not made simultaneous. Whenever a model or language argues that some communication event is to abort because of time-out, it is natural to ask, "Which process timed the time-out?"

This ambiguity about time leads to an ambivalence about time by the more mathematical models and languages. Most of them avoid the subject entirely; the few that mention the problem (such as Shared Variables) discount its significance.

**Table 19-2 Communication**

| System | (D) Synch. | (E) Buffer. | (F) Inform. flow | (G) Commun. control | (H) Commun. Connection send. | rec. |
|---|---|---|---|---|---|---|
| **Shar. Var.** | Asyn. | Bnded. | Bi-Mem | Equal | Port | Port |
| **Ex. Fun.** | Syn. | Bnded. | Bi-Sim | Equal i'm. t.-o. | Port | Port |
| **Con. Proc.** | Syn. | Bnded. | Bi-Sim | Equal | Port | Port |
| **Petri Nets** | Asyn. | Unbnd. | Uni. | Pass. | Port | Port |
| **Data Flow** | Asyn. | Bnded./ Unbnd. | Uni. | Pass. | Name | —— |
| **CSP** | Syn. | Bnded. | Uni. | Equal I/O-grd. | Name pat.-mat. | Name pat.-mat. |
| **Actors** | Asyn. | Unbnd. | Uni. | Pass. | Name | —— |
| **IAP** | Asyn. | Bnded. | Uni. | Equal | Port | Port |
| **Con. Pas.** | Asyn. | Bnded. | Bi-Mem | Equal susp. | Port w/entry | Port w/entry |
| **Dist. Proc.** | Syn. | Bnded. | Bi-Del | Pass. | Entry | —— |
| **Ada** | Syn. | Bnded. | Bi-Del | Act. I/O-grd., time-out | Entry | Entry |
| **PLITS** | Asyn. | Unbnd. | Uni. | Act. send-filt. | Name | Key-filt. |
| **SR** | Syn./ asyn. | Bnded./ Unbnd. | Bi-Del/ Uni. | Act. I-grd., mess-grd., send-filt., prior. | Entry | Entry |
| **Cell** | Syn. | Bnded. | Bi-Del | Act. I-grd., send-filt., prior., susp. | Entry | Entry |
| **Argus** | Syn. | Bnded. | Bi-Del | Pass. | Entry | —— |
| **DH-II** | Asyn. | Unbnd. | Uni. | Equal | Broad. | Pat. mat. |
| **Con. Nets** | Asyn. | Unbnd. | Uni. | Act. | Broad. | Pat. mat. |
| **Ether** | Asyn. | Unbnd. | Uni. | Pass. | Broad. | Pat. mat |

This reflects the attitude that since time is inherently ambiguous any possible formalization is vacuous.

The systems languages, needing to deal with the real world, must take the opposite attitude: "Time may be relative, but if it has been one millisecond

(one second, one minute, one day, ...) since I sent that message and I have not received a reply, the other process is not going to respond."* Only three of the systems have mechanisms for direct temporal manipulation: Exchange Functions's immediate exchange function, Ada's delayed input and output select statements, and the ability of Argus actions to first delay and then kill sibling actions. In some ways, Ether is especially atemporal—sprites have access not only to communications sent after their creation but also to those sent before.

**Fairness (J)** Fairness concerns the bounds on the delay that a process faces before achieving access to a resource. Most of our concerns are with communication fairness, since processes interact through communication. However, there are also fairness considerations involved in processor allocation and peripheral device control. Our systems exhibit three attitudes towards fairness. In an *antifair* system a process can be indefinitely blocked from getting a resource. *Weak fairness* implies that a process is sure to get the resource eventually. *Strong fairness* demands that processes get resources in turn. In a strongly fair system, it is possible to determine a finite upper bound on the number of other requests (of equivalent priority on the given request structure) that will be served before a given request. One common mechanism for enforcing strong fairness is to queue waiting requests. With a queue, a process waits only for those processes ahead of it in line.

Interestingly, there is only a weak correlation between attitudes towards fairness and other dimensions of system organization. Concurrent Processes, Petri Nets, Data Flow with indeterminate-merge,† CSP, Argus, Distributed Hearsay-II, Contract Nets, and Ether are explicitly antifair. Shared Variables, Exchange Functions, IAP, and Actors have weak fairness. For example, in Shared Variables, every process eventually takes another step; in Actors, every message is eventually received and processed. Distributed Processes, Ada, PLITS, SR, and Cell are queue-based systems and are, in some respects, strongly fair.

**Failure (K)** Most models and languages treat processes as fault-free automata and communication as invariably successful. However, several systems have some mechanisms for dealing with failure. With the frons statement, IAP encourages convenient redundancy. Ada has several mechanisms for handling failure and distributed termination, including time-outs and exception handlers. Contract

* The software controlling the first flight of the space shuttle Columbia provided a graphic illustration of the difficulties of programming multiprocessor systems that depend on intricate timing relationships [Garman 81]. Because a central clock would send different processors different times, the programmers of the shuttle controller used an ad hoc arrangement to provide processors with identical times. This system had a bug that, with low probability, allowed the processors to initialize to a permanently unsynchronized state. Unfortunately, when the system was brought up for the planned launch, it fell into this state. This forced a delay of the shuttle flight until the bug was found and fixed.

† Fairness is not an issue in determinate systems.

**Table 19-3 Other issues**

| System | (I) Time | (J) Fairness | (K) Failure | (L) Heuristics |
|---|---|---|---|---|
| **Shar. Var.** | —— | Weak | —— | —— |
| **Ex. Fun.** | Instantaneous time-out | Weak | —— | —— |
| **Con. Proc.** | —— | Anti | —— | —— |
| **Petri Nets** | —— | Anti | —— | —— |
| **Data Flow** | —— | Anti | —— | —— |
| **CSP** | —— | Anti | —— | —— |
| **Actors** | —— | Weak | —— | —— |
| **IAP** | —— | Weak | Convenient redundancy | —— |
| **Con. Pas.** | —— | Anti | —— | —— |
| **Dist. Proc.** | —— | Strong | —— | —— |
| **Ada** | Delayed time-out | Strong | Exception handlers, distributed termination | Distributed termination |
| **PLITS** | —— | Strong | —— | —— |
| **SR** | —— | Strong | —— | —— |
| **Cell** | —— | Strong | —— | —— |
| **Argus** | Delayed time-out | Anti | Atomic actions, exception handlers | Atomic actions |
| **DH-II** | —— | Anti | Evidence | Pattern-directed invocation |
| **Con. Nets** | —— | Anti | Contract managers | Contracts |
| **Ether** | Anti-time | Anti | Evidence | Pattern-directed invocation |

Nets allows the manager of a contract to monitor its progress. Argus provides perhaps the most comprehensive set of conventional failure mechanisms, wrapping each remote call in an atomic action and allowing exception handlers to deal with the failures of these actions. Distributed Hearsay-II and Ether rely on weighing evidence in drawing conclusions, using the natural redundancy of their processing organization to immunize against failure.

**Heuristic Mechanisms (L)** Five systems have specific heuristic mechanisms. Ada provides a primitive for coordinating distributed termination of a set of processes. Argus, extending this idea, provides atomic actions. Contract Nets implements the contract mechanism and negotiation as an organizing principle. And Distributed Hearsay-II and Ether each have a version of pattern-directed invocation for communication.

## 19-2 SPECIFIC COMPARISONS

Many of the themes of particular models and languages are echoed, with variations, in other systems. A set of systems that share several key properties form a family of systems. We find the variations within a family noteworthy. In this section we identify several concepts that define families of systems and note some general metaphors about models and languages.

**Communication metaphors** Human artifacts parallel human experience. One contention of Chapter 18 is that human organizations are candidate models for coordinated systems. We observe that the interprocess communication mechanisms of most of the systems resemble human communication mechanisms. How do people communicate? Face-to-face, direct communication is the most immediate form, but such proximity is the antithesis of distribution. The primary indirect interpersonal communication media are the telephone and the mail. The directness and immediacy of the telephone parallel synchronous communication. For example, in CSP communicating processes "call" each other. However, CSP conversations differ from human telephone calls in two important respects: (1) once attempted, a call cannot be aborted; and (2) only a single message is sent in a conversation. Guarded commands in CSP allow calling several "phones" at once, waiting for the first to answer. Exchange Functions and Concurrent Processes share the telephonic flavor of CSP, though they direct calls to "central switchboards" (ports) instead of using "direct dialing." In Exchange Functions, a process that would otherwise be required to wait for an answer can "hang up."* The asynchronous communication of Actors and PLITS parallels posting letters. One composes a message, drops it in the mail, and continues one's activities.

The other important human communication media are broadcast (like message boards, radio, and television) and archival (like books, records, and newspapers). The communication medium of Shared Variables is like a bulletin board. Messages are written and overwritten, without specifying for whom they are in-

* The synchronous communication of Distributed Processes and its derivatives is unlike this phone metaphor, because the called party can consult arbitrarily many other processes between accepting the call and responding—somewhat like being able to place arbitrarily many callers on hold.

tended. Distributed Hearsay-II and Ether specifically relate their communication organization to blackboards and libraries. No system has explored messages that are generally but only intermittently available.

**Functional and applicative formalisms** Functional languages describe computation as the result of successive applications of functions to input values. Applicative languages, a close cousin of functional languages, emphasize the binding of values to names along with function application. These languages contrast with the assignment operation of imperative languages like Pascal. Five of the systems (Exchange Functions, Concurrent Processes, Data Flow, Actors, and IAP) are in some way functional or applicative. Exchange Functions and Concurrent Processes center on processes with state. They use applicative syntax only to describe the state-succession functions. The other three systems are inspired by applicative and functional programming (and the earliest applicative programming language, pure Lisp). IAP is a direct extension of pure Lisp to include nonevaluating cons and frons and functional objects. Except for the state mechanisms of impure actors, the Actor model is a direct implementation of the lambda calculus. We can easily program nonevaluating cons and frons operators in Actors. In particular, a nonevaluating cons actor receives messages about potential car and cdr fields, remembers the messages, but does not act on them until receiving the corresponding car and cdr requests.

It may strike the reader as odd that we classify Data Flow as an applicative language. After all, the syntax of Data Flow (as we have described it) is graphical. However, Data Flow graphs (without indeterminate-merge) are isomorphic to applicative notation. The concurrency in Data Flow computations is the same as the concurrency provided by parallel argument evaluation in applicative languages. Therefore, Data Flow is, in some sense, just another syntax for applicative programming. From this perspective, Data Flow is distinctive principally for syntactic reasons. Data Flow has a simple syntax for constructing functions with several outputs, allows infinite structures without a special **letrec** construct, and provides strong data typing. Data Flow suffers from the pragmatic disadvantage that the free form of the Data Flow graphs encourages poorly structured programs. Also, unlike IAP, the "push" of Data Flow tokens causes the computation to be done with call-by-value instead of call-by-need.

Indeterminate-merge transforms data flow in much the same way that frons transforms IAP and amb transforms LISP. Chapter 12 discusses the effect of hypothesizing a split operation in suspending cons. Such an operator resembles the inverse of indeterminate-merge. Instead of taking inputs from several lines and merging them into a single stream, split takes a single stream and parcels it out to several lines, feeding each line as it "needs" another token.

What would a Data Flow split operator be like? The split operator of suspending cons presents the next data item to the output line that "needs" one next. But since Data Flow is call-by-value, its split could not know which line "needs" the next output.

**Figure 19-1** Split actor with need lines.

A possible resolution of this difficulty would be to add explicit need inputs to the split actor. Each output line from a split operator would have an associated need input. Split would place a token on that line only if the corresponding need input had a token. In doing so, it would consume the need token. Figure 19-1 shows a split actor with need lines.

Filling split is a possible alternative mechanism for introducing split to Data Flow. Filling split has a single input line and several output lines. It is enabled when there is a token on its input line. On firing, it transfers this token to one of its output lines that does not currently hold a token. Although filling split cannot match split's ability to divide an input stream among several deserving outputs, it can apportion work roughly according to demand among several successors.

**Synchronous models** One family of models is the systems that communicate by immediate, synchronous messages: Exchange Functions, Concurrent Processes, and CSP. These models differ in details: Concurrent Processes allows dynamic process creation, while Exchange Functions and CSP require static processes. Exchange Functions and Concurrent Processes have bidirectional communication through ports, while CSP uses mutual naming modified by pattern matching. Concurrent Processes was defined to investigate the mathematics of concurrent computation; Exchange Functions, to aid in the design of systems. CSP serves a dual purpose, both as a mathematical model and as a systems implementation language. We find it interesting that such diverse domains give rise to such similar systems.

**Perspectives on queue control** When several requests are pending for a particular process, the system must decide which request to handle next. Some systems specify that no particular service order is defined: requests are served arbitrarily. Models tend toward this approach, as they are more mathematical, and antifairness is easier to express mathematically. In particular, CSP and Actors treat all waiting requests equally.

Languages designed for systems development tend toward the opposite extreme. These languages often have elaborate rules for deciding the next request to be processed. These mechanisms express the programming tradition of giving the system implementor maximum control of the computing environment, balanced only by the requirements of straightforward implementation and efficient execution. In our systems languages we see a progression of mechanisms for queue control. Our first such language, Distributed Processes, sorted calls only by their destination procedure. Ada and SR developed this idea of grouping calls, resulting in the concept of task entries, that serve, like monitor procedures, as collectors of requests, but can appear in several places in the program. Entries also began to acquire attributes of their own. In particular, a program can count the requests waiting on an entry.

PLITS, a contemporary of Ada, unified all entries into a single queue but provided the first filtering on that queue: processes can select messages by sender or transaction key. Feldman presented the rationale of that decision as [Feldman 79, p. 359]:

> One would like a module to be able to do quite selective **receive**'s and not be bothered with messages that it was not ready to process. For example, one could allow **receive** to take an arbitrary predicate on the values of slots in the message. There are several difficulties. One cannot build into the system all the generality that might ever be required—for example, a module might want to receive that message that has the greatest value for some slot. Another problem is that having very selective **receive**'s puts a great burden on the system for storing, checking, and keeping track of messages. Finally, there are problems of defining the correct sequencing for messages that are being controlled by complex predicates. The definition we have chosen is a compromise. Clearly, having **receive** only specify the source is too restrictive.

He proceeded to show that many different control regimes can be encoded by transaction keys.

SR adapted the PLITS mechanism of sorting messages by sender and added features of its own: guard and priority clauses based on the values in the message itself. Cell extended these mechanisms with provisions for examining a request and returning it to a suspended state. What Feldman feared as too complex (sorting for the message with the greatest value on some slot) has become the routine of later languages.

What is gained by more complex sorting mechanisms? If the system hides a sorting mechanism, some complicated programs are easily expressed. This is particularly so when one wants the request that is extreme in a way that can be mapped onto the sorting mechanism. In such circumstances, a primitive that selects extremities dramatically simplifies the program. Is SR's approach the ultimate in queue control? Feldman hinted at the answer to this question with his mention of arbitrary predicates. We imagine predicates as objects to be applied to the elements of the queue. Additionally, some criteria are attributes of the queue as a whole. For example, on a given entry, a process might want to select a member of the class of requests with the greatest number of pending calls.

Such a decision involves examining not only the individual elements but also the queue as a whole. Such a control structure is a step beyond the current proposals.

**Asynchronous message systems** Two systems center on asynchronous messages, Actors and PLITS. Though different in appearance (PLITS has an imperative syntax, while Actors is applicative), they are at heart similar. Both systems are based on messages and recognize the importance of being able to pass process names in messages. Both thereby allow a form of continuation. Actors is primarily a model. It provides only a minimal set of primitive mechanisms and rules restricting the behavior of these primitives. PLITS carries the asynchronous message metaphor into practice. It extends this metaphor by adding structure to messages and message receptions. The most important operational difference is that in PLITS, messages sent from the same sender with the same transaction key arrive in the order sent. Actors does not guarantee any such ordering. In practice, the PLITS restriction requires a fairly complicated communication protocol; the Actor perspective may prove more realistic in megacomputing systems. This difference notwithstanding, PLITS can be viewed as a pragmatic, imperative implementation of the actor metaphor.

**Processes** The process concept is basic to most approaches to distributed computing, but the definition of process varies among systems. All processes have permanent storage and the power to compute. In some systems, processes are objects; they can be dynamically allocated and deleted; they possess names that can be passed in messages. In Cell, processes can even express "last wishes" to be executed before they are deallocated. Other systems, for reasons of philosophy or implementation difficulty, do not provide all these facilities. Limitations include fixing the set of processes at compilation time and failing to provide process names, thereby restricting the potential communication structure.

## 19-3 BASIS SYSTEMS

The systems have a variety of mechanisms. We are drawn to the question of natural primitives and desirable extensions: What sets of primitives can describe the behavior of our systems? What mechanisms built from these primitives best support coordinated computing? In this section we discuss these two issues.

### Seeking a Basis

We seek a primitive model, a *basis* model. We want an *operational* model, one that describes what to do without specifying the details of an implementation. We will not impose a syntax on our model, but we want the primitives of the

model to correspond directly to the familiar programming concepts they are meant to emulate.*

Both Milne and Milner and Lynch and Fischer address formal equivalence of concurrent systems. These authors show that a set of concurrent processes is semantically equivalent to a single process with unbounded indeterminacy. Hence, a simple extension of Turing machines will capture the formal semantics of distributed systems. However, we find this formal result unsatisfying; it fails to reflect the operational reality of a distributed environment. Multiple processing agents provide some efficiency advantages, and distribution provides some constraints. We want the basis model to reflect these realities.

Our technique is to go through our list of dimensions, selecting a choice for each dimension that encompasses the others. This results in a basis model; restrictions of this model give us the individual systems discussed in Parts 2, 3, and 4. A key theme of our discussion is protocols—how one system can model another if its processes follow a particular pattern of actions.

Of course, the problem domain of our approach is modeling languages and models. The first real dimension is explicit processes. Almost all the systems use explicit processes. Four systems do not: Petri Nets, Data Flow, IAP, and Distributed Hearsay-II. We can simulate Petri Nets with processes by having a process for each transition and place, Data Flow by having a process for each actor, IAP by creating new processes for each cons or frons, and Distributed Hearsay-II by creating a process for each entry and knowledge source. Thus, explicit processes are adequate to model any of our systems. Some systems allow processes to be (externally) suspended or destroyed. We can model this by requiring processes to check between other communications to see if they should spin or die.

Though a single process with unbounded indeterminacy can simulate any number of other processes, it seems most natural to give our model dynamic process creation. We identified two syntactic mechanisms for creating new processes: lexical elaboration and explicit creation. Clearly, a system that creates new processes by explicit command subsumes lexical elaboration. To model lexical elaboration, we execute "create" instructions in place of process declarations. To model a system of static processes, one merely fails to create any new processes not required by the rest of the model.

Which is more primitive, synchronous or asynchronous communication? We can easily model synchronous communication with asynchronous communication—processes obey a protocol that requires the sender of a message to wait for an acknowledgment and the recipient of a message to send such an acknowledgment. A system with static processes, textually guarded output statements,

* Reid's dissertation [Reid 82] is an attempt similar to this one. Its approach and its conclusions are both more comprehensive and more complex than the theme developed in this section.

and synchronous communication cannot model asynchronous communication. This is because an asynchronous message-based system can create an unbounded number of pending messages, while a synchronous system with static processes cannot have more than one pending message per process per output guard at any time. A system with dynamic process creation and synchronous communication could mimic asynchronous communication by creating a "secretary" process for each message and making that secretary responsible for completing the communication.

We choose asynchronous communication for our model because it is easier to express protocols asynchronously. Along the same lines, we give our model unbounded buffers and unidirectional information flow. Unbounded buffers can certainly model bounded ones; bidirectional information flow can be simulated by a protocol of exchanged messages.

We identified four varieties of interprocess connection: name, entry, port, and broadcast. Typically, a process with a single code point for receiving messages uses names, and a process with several reception points uses entries. Ports are used to target messages that are not necessarily addressed to a particular process. Broadcasting mechanisms distribute a single message to many "appropriate" recipients.

Entries serve to sort and filter messages and to direct messages to particular segments of code. Names and entries are equivalent: a system that uses names can be modeled as a single-entry process, and a system with entries can be modeled as a single name process with a "computed goto" after that name. This goto would jump to the code originally associated with the entry name. This latter modeling requires some facility for filtering requests, such as secretaries or PLITS-like transaction keys.

Ports differ from names and entries in that several processes can share a port. The simplest variety of port receives messages of two kinds: insertions and removals. An insertion adds a message to the port's set of pending messages; a removal deletes a message and responds with that message.

Broadcasting takes several forms. The simplest form of broadcasting is a message directed at a specific finite set of recipients. This is equivalent to a program that loops, sending one message at each step. A second form of broadcasting resembles shared memory—information is available to processes but not channeled directly to them. This can be modeled as a port or, in the unbounded case, as a sequence of ports. A process seeking the latest broadcast could ask a broadcast port and be told both the information itself and the name of the successor broadcast port. The most complicated form of broadcast ties broadcast dispatch with pattern-directed reception. Here, processes specify a message pattern, and the system directs all broadcast messages that match that pattern to the process. We can model this arrangement by giving each process a secretary process that scans all broadcasts and forwards the appropriate ones to its executive.

For the moment, we observe that ports model entries by handling insertion and deletion requests, and that ports model broadcasts by accepting insertions and responding to requests with both information and the names of successor ports. Since ports allow message sharing that is unavailable to entries, we give our basis model ports.*

Communication control presents the widest variety of mechanisms. These mechanisms are used for three purposes: indeterminacy, filters, and time-outs. To handle them, our ports become more active agents — they need programs of their own. Indeterminacy is usually shown by guarded statements. Our port has several ways to select a guard clause. For example, it could choose one at random, survey them in some fixed priority order, or select the earliest arrival. Our port may need a random-number generator to imitate the first mechanism, but in any case, these can all be expressed by programs within the port.†

Programs use filters to select the desired message from among the waiting requests. Typical filters are the message-sender and transaction-key filters in PLITS, the sender and priority filters of SR and Cell, and the pattern-matching of Ether. Of course, if messages are to be filtered by sender, then message senders must decorate messages with their identities. To expect the port to do this filtering requires sending the port a more detailed request. Nevertheless, such filtering is still easily accommodated in our model.

Three systems have time-out mechanisms. Receiver time-outs (Exchange Functions and Ada) require either that the port respond immediately if it is empty (Exchange Functions and Ada conditional select/entry statements) or that it be prepared for a second message from the receiver, telling the port to ignore the initial request (Ada timed select/entry statement). Sender time-outs (Ada and Argus) permit the sender of a message to remove it from the port. Thus, "ports that served as entries" (which we originally presumed would allow only a single process to remove messages) turn out, in certain circumstances, to allow multiple message removers.

Fairness is a subtle issue. Shared Variables provides a convenient formalism for illustrating some of the difficulties in producing a fair implementation. If two processes compete to write a variable, the first may always write just before the second. Thus, the second's writing may prevent any other process from reading the value written by the first. The timing and actions of these two processes may be so regular that this sequencing continues arbitrarily into

* We will later retreat from this assertion back to a simpler name mechanism.

† Two processes can simulate an unbounded random-number generator in the following fashion: the first sends a message asking for a reply to the second process and starts counting. The first process counts until it receives the reply. The usual rules of the temporal independence of processes imply that a sequence of such counts is both unbounded and random.

the future. We can avoid this dilemma by polling — providing each process with a variable that it alone writes and having the message receiver check, in turn, each variable that might contain a message to it. In a dynamic system, a process that created other processes would be responsible for polling them.*

The polling argument shows that if each process continues to make computational progress, strong fairness can be achieved (albeit at the cost of great synchronization). However, the assumption of computational progress is itself an assumption of weak fairness. Thus, achieving strong fairness requires assuming a weakly fair implementation, and a weakly fair implementation can simulate strong fairness.

A weakly fair implementation that wishes to imitate the potential "unfairness" of an antifair system can randomly select particular messages for delayed processing. Hence, weak fairness suffices for imitating the other two kinds of fairness.

What is the computational meaning of time? Good clocks exhibit certain behavior. Specifically, they produce a monotonically increasing sequence of values, where these values have a rough correspondence to both the internal clocks of people and noncomputer clocks. A computer clock thus can be implemented as a counter that is repeatedly incremented. It may be necessary to occasionally adjust the values on this clock to reflect the clocks of the other processes of a system or external clocks. (Lamport has proposed one algorithm for such synchronization [Lamport 78].) Therefore, we see that temporal constructs do not have to be primitive in the model, but can be simulated by counting, "clock" processes.

Similarly, each of the mechanisms for dealing with failure is an algorithm relying on more primitive events. When presenting a system with failure mechanisms, the authors take pains to describe the algorithms of the failure mechanisms in terms of more basic notions. In particular, the most fundamental failure notion is that of failure to receive a reply within a specified time. The basis model can simulate these time-outs with alarm processes.

Our ports have progressed from simple message repositories to elaborate message handlers, complete with facilities for storing, sorting, and organizing messages. Thus, the ports themselves have become processes. Who handles messages for our port/processes? It is inappropriate to imagine, for this port/process, a port/process of its own. This would create an infinite regress of port/processes. Instead, let the port/process handle the messages that come to it in a first-come, first-served order, at a single entry point. With this observation we see that the port/process is just an ordinary process. We have eliminated the need for a

* In his paper on time and clocks, Lamport presents an algorithm for ensuring strong fairness [Lamport 78]. Unfortunately, his algorithm requires a broadcast to all other processes at each significant event.

special variety of port. Instead, processes that require a special kind of filtering declare a secretary process to do that filtering for them and have their messages directed to the secretary.

In summary, our basis has dynamic creation of explicit processes. These processes communicate asynchronously, receive messages at a single entry, and enforce weak fairness. It has little else in the way of queue organization. We have shown how this organization is primitive and natural; it can directly model each of the other models and languages.

## Ideal Language and Heuristic Systems

The discussion of the basis model argued that the appropriate set of primitives could model any of our systems. In this section we consider the opposite side of the design question: Which macro operations, built from these primitives, should an "ideal" system provide? Once again, our discussion parallels our dimensions.

In one sense, such an ideal system leaves the programmer unaware of the lower-level concepts of programming, such as the existence of processes. Instead, the programmer describes tasks, and the system arranges its resources to carry out those tasks concurrently. This resembles the goals of the heuristic systems of Chapter 18. Those proposals are clearly heading toward the successors of programming languages. However, the systems described in that section represent only the infancy of that technology.

Though it may be pleasant to contemplate the demise of explicit processes, we recognize that processes are the mainstream of current distributed language design. If we take it as given that our system must resemble a language, what features do we give this "ideal language"? Clearly, we want it to have the flexibility that comes from dynamic process creation. Asynchronous communication is (in a naive sense) more general than synchronous, but many process requests require an answer, much as many procedure calls require that a value be returned. The syntax of our language should provide a macro operation (send and immediate receive) to express this variety of interaction. Similarly, the multiple choice of guarded commands should expand into a "multiple send, receive the first answer, and cancel the remaining requests." In an inherently asynchronous system, information flow at the primitive level is unidirectional. These macros build bidirectional information flow into the system.

Naming and access mechanisms mark communication control. Providing processes with explicit entries serves a preliminary sorting function. The various selection mechanisms (filters, guards) are all variations of programs that run through the current input queue to select a particular element. The resolution of this theme is to turn the entries into explicit objects that respond to insertions and comparisons. Transaction and priority filters and guards then become simple programs over these queues. That is, we observe that clever ports

can simulate filter and control mechanisms and provide clever ports as part of the language.

So far our ideal language is notably similar to our basis model. We diverge with the items of our third table — time, failure, and fairness. The queueing mechanism of the ideal language encourages a stronger form of fairness than weak fairness, though we do not demand a fairness that corresponds with external time.

Internal time should be implemented with explicit clock processes. A process that wishes to time-out a request should be able to do so by sending a "withdrawal" request to the destination port. If the port has not yet sent the request to the destination process, it should remove that message from the queue and return an acknowledgment to the original requestor.

A process can recognize a failure in one of two ways. Either a process explicitly fails (like the **self destruct** statement in PLITS) or failure can be inferred from the lack of response to a request. Explicit failure is a variety of message that a process sends in response to a request. It can be so encoded. Failure from a lack of response arises when a request does not receive a response within some program-determined duration. Such a limit should be potentially explicit in every request that demands a response. The concurrent subactions of Argus provide a particularly elegant integration of failure and time-outs. The ideal language should provide a library of such algorithms, not limited to atomicity, but also including the portlike processes discussed above and the resource allocation algorithms discussed in Chapter 18. Textually, such a language must simplify the task of interrupting the flow of program control to handle unusual messages, much as exception handlers in conventional languages deal with system exceptions. Our ideal language will profit if it is able to treat its program state as an object to be manipulated, delayed, and resumed on request.

## PROBLEMS

**19-1** Identify another common theme in the systems studied and analyze the perspectives on that theme.

**19-2** Design a distributed language or model. Analyze your system in terms of the dimensions discussed in Section 19-1.

**19-3** Design a distributed language or model by "making a choice" for each of the columns of Tables 19-1, 19-2 and 19-3. Besides syntax, what else do you need to specify for your system?

† **19-4** Implement your language or model design from Exercise 19-3.

**19-5** To what extent can the effect of filling split be imitated with an indeterminate-merge, a demultiplexer, and an (implicit) queue? What are the limitations of such a solution?

**19-6** To what extent does the basis model resemble Actors? How and where does it differ?

**19-7** The previous question suggests that an Actor-like system can model the other systems. Select another one of the systems described in this book and show that that system can serve as a basis for the others.

**19-8** Which systems cannot serve as a basis, and why not?

## REFERENCES

[**Andrews 83**] Andrews, G. R., and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *Comput. Surv.*, vol. 15, no. 1 (March 1983), pp. 3–43. Andrews and Schneider identify several important linguistic mechanisms for concurrency control and relate several distributed and concurrent languages to these mechanisms.

[**Arvind 77**] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," in B. Gilchrist (ed.), *Information Processing 77: Proceedings of the IFIP Congress 77*, North Holland, Amsterdam (1977), pp. 849–854.

[**Dennis 74**] Dennis, J. B., "First Version of a Data Flow Procedure Language," in B. Robinet (ed.), *Proceedings, Colloque sur la Programmation*, Lecture Notes in Computer Science 19, Springer-Verlag, Berlin (1974), pp. 362–376.

[**Feldman 79**] Feldman, J. A., "High Level Programming for Distributed Computing," *CACM*, vol. 22, no. 6 (June 1979), pp. 353–368.

[**Filman 80**] Filman, R. E., and D. P. Friedman, "Inspiring Distribution in Distributed Computing," *Working Papers ACM SIGOPS/SIGPLAN Workshop Fundam. Issues Distrib. Comput.*, Fallbrook, California (December 1980), pp. 53–59. Also available as Technical Report 99, Computer Science Department, Indiana University, Bloomington, Indiana (December 1980).

[**Filman 82**] Filman, R. E., and D. P. Friedman, "Models, Languages, and Heuristics for Distributed Computing," *1982 National Computer Conference*, AFIPS Conference Proceedings vol. 51, AFIPS Press, Arlington, Virginia (1982), pp. 671–678.

[**Garman 81**] Garman, J. R., "The 'Bug' Heard 'Round the World," *Softw. Eng. Notes*, vol. 6, no. 5 (October 1981), pp. 3–10. This paper presents an intriguing example of the complexity of time in multiprocessor systems.

[**Lamport 78**] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, vol. 21, no. 7 (July 1978), pp. 558–565. Lamport discusses the meaning of event ordering in a distributed system. He relates the timing of events in separate processes to the notion of relativistic time, provides a theory of distributed ordering, and gives an algorithm for strong fairness. Unfortunately, his algorithm requires broadcast communication to all processes at each event.

[**Mohan 80**] Mohan, C., "A Perspective of Distributed Computing: Models, Languages, Issues and Applications," Working Paper DSG-8001, Department of Computer Science, The University of Texas, Austin, Texas (March 1980). Mohan identifies several design criteria for language design and classifies 16 languages and systems by these criteria.

[**Rao 80**] Rao, R., "Design and Evaluation of Distributed Communication Primitives," Technical Report 80-04-01, Department of Computer Science, University of Washington, Seattle, Washington (April 1980). Rao discusses communication primitives for distributed programming. He gives an overview of the communication perspectives of several languages and language proposals, and presents an example of a sorting program in each.

[**Reid 82**] Reid, L. G., *Control and Communication in Programs*, UMI Research Press, Ann Arbor (1982). Reid develops a model of interprocess communication and proves several theorems about the various equivalences of the model. Her model is based on explicit, uniquely named processes that communicate over named identifiers associated with those processes (what she call "ports"). There are operations to connect and disconnect the ports of different processes and to communicate information between processes, both synchronously and asynchronously.

[**Stotts 82**] Stotts, P. D., "A Comparative Study of Concurrent Programming Languages," *SIGPLAN Not.*, vol. 17, no. 10 (October 1982), pp. 50–61. Stotts applies a dimensional analysis in studying several concurrent (and distributed) languages.